Merging Nodes in Search Trees: an Exact Exponential Algorithm for the Single Machine Total Tardiness Scheduling Problem[*]

Lei Shang[1], Michele Garraffa[2], Federico Della Croce[3], and Vincent T'Kindt[4]

1 Université François Rabelais de Tours, Laboratoire d'Informatique (EA 6300), ERL CNRS OC 6305, Tours, France
   shang@univ-tours.fr
2 Politecnico di Torino, DAUIN, Torino, Italy
   michele.garraffa@polito.it
3 Politecnico di Torino, DIGEP, Torino, Italy
   federico.dellacroce@polito.it
4 Université François Rabelais de Tours, Laboratoire d'Informatique (EA 6300), ERL CNRS OC 6305, Tours, France
   tkindt@univ-tours.fr

Abstract

This paper proposes an exact exponential algorithm for the problem of minimizing the total tardiness of jobs on a single machine. It exploits the structure of a basic branch-and-reduce framework based on the well known Lawler's decomposition property. The proposed algorithm, called branch-and-merge, is an improvement of the branch-and-reduce technique with the embedding of a node merging operation. Its time complexity is $\mathcal{O}^*(2.247^n)$ keeping the space complexity polynomial. The branch-and-merge technique is likely to be generalized to other sequencing problems with similar decomposition properties.

1   Introduction

The challenge of designing exact exponential algorithms for NP-hard problems is attracting more and more researchers, particularly since the beginning of this century. For a survey on the most effective techniques in designing exact exponential algorithms, readers are kindly referred to Woeginger's paper [13] and to the book by Fomin and Kratsch [4]. In spite of the growing interest on exact exponential algorithms, few results are yet known on scheduling problems, see the survey of Lenté et al. [8].[1] This paper focuses on a pure sequencing problem, the single machine total tardiness problem, denoted by $1||\sum T_j$. In this problem, a

---

[*] A full version of the paper is available at [5], https://hal.archives-ouvertes.fr/hal-01477835.
[1] Recent results on Parameterized Algorithms of scheduling problems can be found at http://fpt.wikidot.com/operations-research.

jobset $N = \{1, 2, \ldots, n\}$ of n jobs must be scheduled on a single machine. For each job j, a processing time p_j and a due date d_j are given. The problem asks for arranging the jobset in a sequence S so as to minimize $T(N, S) = \sum_{j=1}^n T_j = \sum_{j=1}^n \max\{C_j - d_j, 0\}$, where C_j is the completion time of job j in sequence S. The $1||\sum T_j$ problem is NP-hard in the ordinary sense [2]. It has been extensively studied in the literature. The current state-of-the-art exact method [11] solves to optimality problems with up to 500 jobs. Its complexity was not discussed in [11] but will be analyzed in this paper. In [7] an exact pseudo-polynomial dynamic programming algorithm was proposed with complexity $\mathcal{O}(n^4 \sum p_i)$. Also, the standard technique of doing dynamic programming across the subsets (see, for instance, [4]) applies and runs with complexity $\mathcal{O}(n^2 2^n)$ both in time and in space. We refer to [6] for a comprehensive survey on the problem. In the rest of the paper, the $\mathcal{O}^*(\cdot)$ notation [13], commonly used in the context of exact exponential algorithms, is used. Let $T(\cdot)$ be a super-polynomial and $p(\cdot)$ be a polynomial, both on integers. In what follows, for an integer n, we express running-time bounds of the form $\mathcal{O}(p(n) \cdot T(n)))$ as $\mathcal{O}^*(T(n))$. As an example, the complexity of dynamic programming across the subsets for the total tardiness problem can be expressed as $\mathcal{O}^*(2^n)$. The aim of this work is to design a faster exact exponential algorithm running in $\mathcal{O}^*(c^n)$ (c being a constant) and polynomial space, exploiting known decomposition properties of the problem. The designed algorithm, making use of a new technique called branch-and-merge that avoids the solution of several equivalent subproblems in the branching tree, is shown to have a complexity $\mathcal{O}^*(2.247^n)$ in time and requires polynomial space. We also provide a complexity analysis of the state-of-the-art exact algorithm [11], which runs in $\mathcal{O}^*(2.4143^n)$ in time.

2 Preliminaries

We recall some basic properties of the total tardiness problem and related notation. Given a jobset $N = \{1, 2, \ldots, n\}$, let $(1, 2, \ldots, n)$ be a LPT (Longest Processing Time first) sequence, where $i < j$ whenever $p_i > p_j$ (or $p_i = p_j$ and $d_i \leq d_j$). Let also $([1], [2], \ldots, [n])$ be an EDD (Earliest Due Date first) sequence, where $i < j$ whenever $d_{[i]} < d_{[j]}$ (or $d_{[i]} = d_{[j]}$ and $p_{[i]} \leq p_{[j]}$). The order of jobs having identical processing time and due date should be fixed arbitrarily. Jobs are processed with no interruption starting from time zero. Let B_j and A_j be the sets of jobs that precede and follow job j in an optimal sequence being constructed. Correspondingly, the completion time of job j, $C_j = \sum_{k \in B_j} p_k + p_j$. Also, if job j is assigned to position k, $C_j(k)$ denotes the corresponding completion time and $B_j(k)$ and $A_j(k)$ the sets of predecessors and successors of j, respectively. The following known theoretical properties hold.

▶ **Property 1.** *[3] Consider two jobs i and j with $p_i < p_j$. Then, in at least one optimal schedule, i precedes j if $d_i \leq \max\{d_j, C_j\}$, otherwise j precedes i if $d_i + p_i > C_j$.*

▶ **Property 2.** *[7] Let job 1 in LPT order correspond to job $[k]$ in EDD order. Then, job 1 can be set only in positions $h \geq k$ and the jobs preceding and following job 1 are uniquely determined as $B_1(h) = \{[1], [2], \ldots, [k-1], [k+1], \ldots, [h]\}$ and $A_1(h) = \{[h+1], \ldots, [n]\}$.*

▶ **Property 3.** *[7, 9, 10] Consider $C_1(h)$ for $h \geq k$. Job 1 cannot be set in position $h \geq k$ if:*
(a) $C_1(h) \geq d_{[h+1]}$, $h < n$;
(b) $C_1(h) < d_{[r]} + p_{[r]}$, for some $r = k+1, \ldots, h$.

▶ **Property 4.** *([12]) For any pair of adjacent positions $(i, i+1)$ that can be assigned to job 1, at least one of them is eliminated by Property 3.*

Algorithm 1 Total Tardiness Branch-and-Reduce (TTBR)

Input: $N = \{1, ..., n\}$ is the problem to be solved
1: **function** TTBR(S, t)
2:     $seqOpt \leftarrow$ a random sequence of jobs
3:     $l \leftarrow$ the longest job in N
4:     **for** $i = 1$ to n **do**
5:         Branch by assigning job l to position i if not discarded by Property 3
6:         $seqLeft \leftarrow$ TTBR$(B_l(i), t)$
7:         $seqRight \leftarrow$ TTBR$(A_l(i), t + \sum_{k \in B_l(i)} p_k + p_l)$
8:         $seqCurrent \leftarrow$ concatenation of $seqLeft$, l and $seqRight$
9:         $seqOpt \leftarrow$ best solution between $seqOpt$ and $seqCurrent$
10:     **end for**
11:     **return** $seqOpt$
12: **end function**

A basic branch-and-reduce algorithm TTBR (Total Tardiness Branch-and-Reduce) can be designed by exploiting Property 2, which allows to decompose the problem into two smaller subproblems when the position of the longest job l is given and by taking into account Property 4 which states that for each pair of adjacent positions $(i, i+1)$, at least one of them can be discarded. The basic idea is to iteratively branch by assigning job l to every possible position $\{1, ..., n\}$, discarding ineligible positions by means of the elimination rules of Property 3, and correspondingly decompose the problem. Each time a certain position i is selected for job l, two different subproblems are generated, corresponding to schedule the jobs before l (inducing subproblem $B_l(i)$) and after l (inducing subproblem $A_l(i)$), respectively. The algorithm operates by applying to any given jobset S starting at time t function $TTBR(S, t)$ that computes the corresponding optimal solution. With this notation, the original problem is indicated by $N = \{1, ..., n\}$ and the optimal solution is reached when function $TTBR(N, 0)$ is computed. The algorithm proceeds by solving the subproblems along the branching tree according to a depth-first strategy and runs until all the leaves of the search tree have been reached. Finally, it provides the best solution found as an output. Algorithm 1 summarizes the structure of this approach, while Proposition 5 states its worst-case complexity.

▶ **Proposition 5.** *Algorithm TTBR runs in $\mathcal{O}^*((1 + \sqrt{2})^n) = \mathcal{O}^*(2.4143^n)$ time and polynomial space in the worst case.*

Proof. We refer to problems where n is odd, but the analysis for n even is substantially the same. Whenever the longest job 1 is assigned to the first and the last position of the sequence, two subproblems of size $n - 1$ are generated. For each $2 \leq i \leq n - 1$, two subproblems with size $i - 1$ and $n - i$ are generated. Hence, the total number of generated subproblems is at most $2n - 2$. This would induce the following recurrence for the running time $T(n)$:

$$T(n) = 2T(n-1) + 2T(n-2) + ... + 2T(2) + 2T(1) + \mathcal{O}(p(n)) \tag{1}$$

However, Property 4 indicates that the elimination rules of Property 3 discard at least one position for every pair of adjacent positions. The worst case occurs when the largest possible subproblems are kept that is when subproblems with size $n - 1, n - 3, n - 5, \ldots$ (that arise by branching on positions i and $n - i + 1$ with i odd) are kept and correspondingly subproblems with size $n - 2, n - 4, n - 6, \ldots$ are discarded. This induces a recurrence of the type:

$$T(n) = 2T(n-1) + 2T(n-3) + ... + 2T(4) + 2T(2) + \mathcal{O}(p(n)) \tag{2}$$

By replacing n with $n - 2$, the following expression is derived:

$$T(n - 2) = 2T(n - 3) + 2T(n - 5) + ... + 2T(4) + 2T(2) + \mathcal{O}(p(n - 2)) \tag{3}$$

Plugging expression 3 into expression 2, we get:

$$T(n) = 2T(n - 1) + T(n - 2) + \mathcal{O}(p(n)) \tag{4}$$

that induces as complexity $\mathcal{O}^*((1 + \sqrt{2})^n) = \mathcal{O}^*(2.4143^n)$. The space requirement is polynomial since the branching tree is explored according to a depth-first strategy.   ◄

The current state-of-the-art algorithm described in [11], noted hereafter as BB2001, is a branch and bound algorithm having a similar structure as that of TTBR. The main difference is that in BB2001, besides of the decomposition rule given in Property 2, another decomposition rule, based on Property 6, is applied simultaneously on each branching. We provide in Proposition 7 our analysis on the time complexity of BB2001, since this is not discussed in [11]. Notice that even though the time complexity of TTBR is the same as BB2001, the former one serves as a basis of the final algorithm branch-and-merge.

▶ **Property 6.** *[1] Let job k in LPT sequence correspond to job $[1]$ in EDD sequence. Then, job k can be set only in positions $h \leq (n - k + 1)$ and the jobs preceding job k are uniquely determined as $B_k(h)$, where $B_k(h) \subseteq \{k + 1, k + 2, ..., n\}$ and $\forall i \in B_k(h), j \in \{n, n - 1, ..., k + 1\} \setminus B_k(h), d_i \leq d_j$*

▶ **Proposition 7.** *Algorithm BB2001 runs in $\mathcal{O}^*(2.4143^n)$ time and polynomial space in the worst case.*

Proof. Before branching on a node, BB2001 first computes the possible positions for the longest job and the job with smallest due date. Then a new branch is created by assigning a pair of compatible positions to these two jobs. We consider two cases as follows.

Firstly, consider the case where job $1 = [n]$. The two decomposition rules become identical and if this condition is also verified in all subproblems, then the time complexity is $\mathcal{O}^*(2.4143^n)$ as proved in Proposition 5.

In the case where $1 \neq [n]$, the worst case occurs when $1 = [2]$ and $[n] = 2$, since in this case we have maximum available branching positions: job $[n]$ can be branched on position $i \in \{1, ..., n - 1\}$ and job 1 can be branched on position $j \in \{2, ..., n\}$, with $i < j$ for each branching. Moreover, we recall that the Property 4 is also valid.

Three subproblems (left, middle and right) are created on each double branching (zero-sized problems are counted). For the sake of simplicity, we note $T(l, m, r) = T(l) + T(m) + T(r)$.

The following recurrence holds.

$$T(n) = \sum_{\substack{i=1 \\ i \ is \ odd}}^{n-1} \sum_{\substack{j=i+1 \\ j \ is \ even}}^{n} (T(i-1, j-i-1, n-j)) + \mathcal{O}(p(n)) \tag{5}$$

$$= T(0, 0, n-2) + T(0, 2, n-4) + T(0, 4, n-6) + ... + T(0, n-2, 0) + \tag{6}$$

$$T(2, 0, n-4) + T(2, 2, n-6) + ... + T(2, n-4, 0) + \tag{7}$$

$$... \tag{8}$$

$$T(n-4, 0, 2) + T(n-4, 2, 0) + \tag{9}$$

$$T(n-2, 0, 0) + \tag{10}$$

$$\mathcal{O}(p(n)) \tag{11}$$

$$= 3 * (T(n-2) + 2T(n-4) + 4T(n-6) + ... + \frac{n}{2}T(0)) \tag{12}$$

$$\tag{13}$$

By applying a similar process of simplification as in the proof of Proposition 5, the following result is finally derived:

$$T(n) = 5T(n-2) - T(n-4). \tag{14}$$

Correspondingly, we have $T(n) = \mathcal{O}^*(\sqrt{\frac{5+\sqrt{21}}{2}}^n) = \mathcal{O}^*(2.1890^n)$. Therefore the worst case occurs when the two decomposition rules overlap, and the resulting time complexity is the same as TTBR, namely $\mathcal{O}^*(2.4143^n)$.

In terms of space complexity, BB2001 applies an extra technique called *Memorization* which makes use of exponential memory space for accelerating the solution. When this extra technique is not considered, the space complexity of BB2001 is also polynomial since depth-first exploration is adopted. ◀

3 Merging nodes in the search tree

In this section, we describe how to get an algorithm running with complexity $\mathcal{O}^*(2.247^n)$ in time and polynomial space by integrating a node-merging procedure into TTBR. The resulting algorithm will be called *branch-and-merge*. We recall that in TTBR the branching scheme is defined by assigning the longest unscheduled job to each available position and accordingly divide the problem into two subproblems. To facilitate the description of the algorithm, we focus on the worst-case scenario where the LPT sequence $(1, ..., n)$ coincides with the EDD sequence $([1], ..., [n])$: in this case no position can be eliminated by Property 2 at each branching.

Figure 1 shows how an input problem $\{1, ..., n\}$ is decomposed by the branching scheme of TTBR. Each node is labelled by the corresponding subproblem P_j (P denotes the input problem) and it is assumed in this example that Property 3 is not applied (for convenience purpose). Notice that from now on $P_{j_1, j_2, ..., j_k}, 1 \leq k \leq n$, denotes the problem (node in the search tree) induced by the branching scheme of TTBR when the largest processing time job 1 is in position j_1, the second largest processing time job 2 is in position j_2 and so on till the k-th largest processing time job k being placed in position j_k.

To roughly illustrate the guiding idea of the merging technique introduced in this section, consider Figure 1. Noteworthy, nodes P_2 and $P_{1,2}$ are identical except for the initial subsequence (21 vs 12). This fact implies, in this particular case, that the problem of
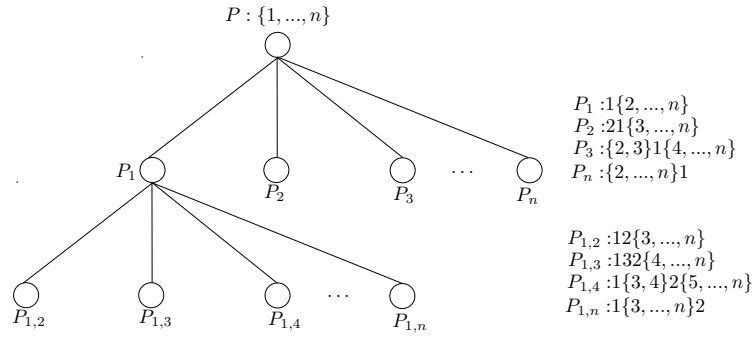
Figure 1 The branching scheme of TTBR at the root node.

scheduling jobset $\{3, ..., n\}$ at time $p_1 + p_2$ is solved twice. This kind of redundancy can however be eliminated by merging node P_2 with node $P_{1,2}$ and creating a single node in which the best sequence among 21 and 12 is scheduled at the beginning and the jobset $\{3, ..., n\}$, starting at time $p_1 + p_2$, remains to be branched on. Furthermore, the best subsequence (starting at time $t = 0$) between 21 and 12 can be computed in constant time. Hence, the node created after the merging operation involves a constant time preprocessing step plus the search for the optimal solution of jobset $\{3, ..., n\}$ to be processed starting at time $p_1 + p_2$. We remark that, in the branching scheme of TTBR, for any constant $k \geq 3$, the branches corresponding to P_i and P_{n-i+1}, with $i = 2, ..., k$, are decomposed into two problems where one subproblem has size $n - i$ and the other problem has size $i - 1 \leq k$. Correspondingly, the merging technique presented on problems P_2 and $P_{1,2}$ can be generalized to all branches inducing problems of sizes less than k. Notice that, by means of algorithm TTBR, any problem of size less than k requires, to be solved, at most $\mathcal{O}^*(2.4143^k)$ time (that is constant time when k is fixed). In the remainder of the paper, for any constant k, we denote by left-side branches the search tree branches corresponding to problems $P_1, ..., P_k$.

With respect to algorithm TTBR, the basic idea is to applying merging on the left-side branches (nodes P_1 to P_k) while Property 3 is applied on the remaining branches (nodes P_{k+1} to P_n).

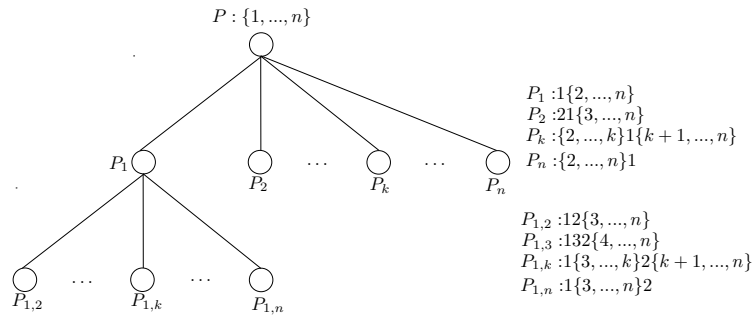## 3.1 Merging left-side branches

We first illustrate the merging operations at the root node. The following lemma highlights two properties of the pairs of problems P_j and $P_{1,j}$ with $2 \leq j \leq k$.

▶ **Lemma 8.** *For a pair of problems P_j and $P_{1,j}$ with $2 \leq j \leq k$, the following conditions hold:*

1. *The solution of problems P_j and $P_{1,j}$ involves the solution of a common subproblem which consists in scheduling jobset $\{j + 1, ..., n\}$ starting at time $t = \sum_{i=1,...,j} p_i$.*
2. *Both in P_j and $P_{1,j}$, at most k jobs have to be scheduled before jobset $\{j + 1, ..., n\}$.*

Proof. As problems P_j and $P_{1,j}$ are respectively defined by $\{2, ..., j\}1\{j + 1, ..., n\}$ and $1\{3, ..., j\}2\{j + 1, ..., n\}$, the first part of the property is straightforward.

The second part can be simply established by counting the number of jobs to be scheduled before jobset $\{j+1, ..., n\}$ when j is maximal, *i.e.* when $j = k$. In this case, jobset $\{k+1, ..., n\}$ has $(n - k)$ jobs which implies that k jobs remain to be scheduled before that jobset.   ◀

(a) Left-side branches of P before performing the merging operations.
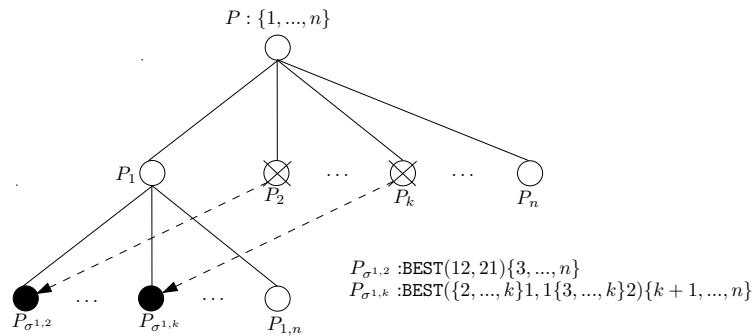
(b) Left-side branches of P after performing the merging operations.

Figure 2 Left-side branches merging at the root node.

Each pair of problems indicated in Lemma 8 can be merged as soon as they share the same subproblem to be solved. More precisely, $(k-1)$ problems P_j (with $2 \leq j \leq k$) can be merged with the corresponding problems $P_{1,j}$.

Figure 2 illustrates the merging operations performed at the root node. For any given $2 \leq j \leq k$, problems P_j and $P_{1,j}$ share the same subproblem $\{j+1, ..., n\}$ starting at time $t = \sum_{i=1}^{j} p_i$. Hence, by merging the left part of both problems which is constituted by jobset $\{1, ..., j\}$ having size $j \leq k$, we can delete node P_j and replace node $P_{1,j}$ in the search tree by the node $P_{\sigma^{1,j}}$ which is defined as follows (Figure 2b):

- Jobset $\{j+1, ..., n\}$ is the set of jobs on which it remains to branch.
- Let $\sigma^{1,j}$ be the sequence of branching positions on which the j longest jobs $1, ..., j$ are branched, that leads to the best jobs permutation between $\{2, ..., j\}1$ and $1\{3, ..., j\}2$ when these two are solved. This involves the solution of two problems of size at most $k-1$ (in $\mathcal{O}^*(2.4143^k)$ time by TTBR) and the comparison of the total tardiness value of the two sequences obtained.

In the following, we describe how to apply analogous merging operations on any node of the tree. With respect to the root node, the only additional consideration is that the children nodes of a generic node may have already been concerned by previous merging operations. Let us refer to `LEFT_MERGE` as the procedure which, for any node of the search tree, perform merging operations on its leftmost child branches. The `LEFT_MERGE` procedure operates based on a modified branching scheme, with respect to TTBR.

Let $\mathcal{L}_\sigma$ be a data structure associated to a problem P_σ. It represents a list of $k-1$ subproblems that result from a previous merging and are now the first $k-1$ children nodes of P_σ. When P_σ is created by branching, $\mathcal{L}_\sigma = \emptyset$. When a merging operation sets the first

$k-1$ children nodes of P_σ to $P_{\sigma^1}, ..., P_{\sigma^{k-1}}$, we set $\mathcal{L}_\sigma = \{P_{\sigma^1}, ..., P_{\sigma^{k-1}}\}$. As a conclusion, the following branching scheme for a generic node of the tree holds.

▶ **Definition 9.** The branching scheme for a generic node P_σ is defined as follows:

- If $\mathcal{L}_\sigma = \emptyset$, use the branching scheme of TTBR;
- If $\mathcal{L}_\sigma \neq \emptyset$, extract problems from $\mathcal{L}_\sigma$ as the first $k-1$ branches, then branch on the longest job in the available positions from the k-th to the last according to Property 2.

This branching scheme, whenever necessary, will be referred to as **improved branching**.

Before describing how merging operations can be applied on a generic node P_σ, we highlight its structural properties by means of Proposition 10.

▶ **Proposition 10.** *Let P_σ be a problem to branch on, and σ be the permutation of positions assigned to jobs $1, \ldots, |\sigma|$, with σ empty if no positions are assigned. The following properties hold:*

1. *$j^* = |\sigma| + 1$ is the job to branch on,*
2. *j^* can occupy in the branching process, positions $\{\ell_b, \ell_b + 1, \ldots, \ell_e\}$, where*

$$\ell_b = \begin{cases} |\sigma| + 1 & \text{if } \sigma \text{ is a permutation of } 1, \ldots, |\sigma| \text{ or } \sigma \text{ is empty} \\ \rho_1 + 1 & \text{otherwise} \end{cases}$$

with $\rho_1 = \max\{i : i > 0, \text{ positions } 1, \ldots, i \text{ are in } \sigma\}$ and

$$\ell_e = \begin{cases} n & \text{if } \sigma \text{ is a permutation of } 1, \ldots, |\sigma| \text{ or } \sigma \text{ is empty} \\ \rho_2 - 1 & \text{otherwise} \end{cases}$$

with $\rho_2 = \min\{i : i > \rho_1, i \in \sigma\}$

Proof. According to the definition of the notation P_σ, σ is a sequence of positions that are assigned to the longest $|\sigma|$ jobs. Since we always branch on the longest unscheduled job, the first part of the proposition is straightforward. The second part aims at specifying the range of positions that job j^* can occupy. Two cases are considered depending on the content of σ:

- If σ is a permutation of $1, \ldots, |\sigma|$, it means that the longest $|\sigma|$ jobs are set on the first $|\sigma|$ positions, which implies that the job j^* should be branched on positions $|\sigma| + 1$ to n
- If σ is not a permutation of $\{1, \ldots, |\sigma|\}$, it means that the longest $|\sigma|$ jobs are not set on consecutive positions. As a result, the current unassigned positions may be split into several ranges. As a consequence of Property 2, the longest job j^* should necessarily be branched on the first range of free positions, that goes from ρ_1 to ρ_2. Under the worst-case scenario, let us consider as an example $P_{1,9,2,8}$, whose structure is $13\{5, \ldots, 9\}42\{10, \ldots, n\}$ and the job to branch on is 5. In this case, we have: $\sigma = (1, 9, 2, 8)$, $\ell_b = 3$, $\ell_e = 7$. It is easy to verify that 5 can only be branched on positions $\{3, \ldots, 7\}$ since 5 must stay before 4 as a direct result of Property 2. ◀

Corollary 11 emphasizes the fact that even though a node may contain several ranges of free positions, only the first range is the current focus since we only branch on the longest job in eligible positions.

▶ **Corollary 11.** *Problem P_σ has the following structure:*

$$\pi\{j^*, \ldots, j^* + \ell_e - \ell_b\}\Omega$$

with π the subsequence of jobs on the first $\ell_b - 1$ positions in σ and Ω the remaining subset of jobs to be scheduled after position ℓ_e (some of them can have been already scheduled). The merging procedure is applied on jobset $\{j^*, \ldots, j^* + \ell_e - \ell_b\}$ starting at time $t_\pi = \sum_{i \in \Pi} p_i$ where Π is the jobset of π.
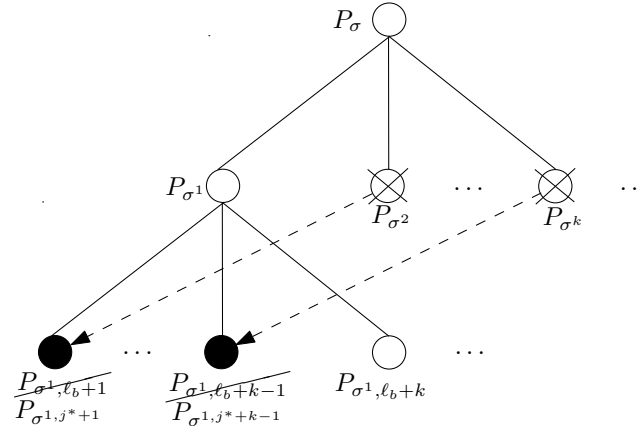
Figure 3 Merging for a generic left-side branch.

The validity of merging on a general node still holds as indicated in Proposition 12, which extends the result stated in Proposition 8.

▶ **Proposition 12.** *Let P_σ be a generic problem and let $\pi, j^*, \ell_b, \ell_e, \Omega$ be computed relatively to P_σ according to Corollary 11. If $\mathcal{L}_\sigma = \emptyset$ the j-th child node P_{σ^j} is $P_{\sigma, \ell_b + j - 1}$ for $1 \leq j \leq k$. Otherwise, the j-th child node P_{σ^j} is extracted from $\mathcal{L}_\sigma$ for $1 \leq j \leq k-1$, while it is created as $P_{\sigma, \ell_b + k - 1}$ for $j = k$. For any pair of problems P_{σ^j} and $P_{\sigma^1, \ell_b + j - 1}$ with $2 \leq j \leq k$, the following conditions hold:*

1. *Problems P_{σ^j} and $P_{\sigma^1, \ell_b + j - 1}$ with $2 \leq j \leq k$ have the following structure:*
   - P_{σ^j}:

   $$\begin{cases} \pi^j \{j^*+j, \ldots, j^*+\ell_e-\ell_b\}\Omega & 1 \leq j \leq k-1 \text{ and } \mathcal{L}_\sigma \neq \emptyset \\ \pi\{j^*+1, \ldots, j^*+j-1\}j^*\{j^*+j, \ldots, j^*+\ell_e-\ell_b\}\Omega & \begin{array}{l}(1 \leq j \leq k-1; \mathcal{L}_\sigma = \emptyset) \\ \text{or } j=k\end{array} \end{cases}$$

   - $P_{\sigma^1, \ell_b + j - 1}$:
     $\pi^1\{j^*+2, \ldots, j^*+j-1\}(j^*+1)\{j^*+j, \ldots, j^*+\ell_e-\ell_b\}\Omega$

2. *By solving all the problems of size less than k, that consist in scheduling the jobset $\{j^*+1, \ldots, j^*+j-1\}$ between π and j^* and in scheduling $\{j^*+2, \ldots, j^*+j-1\}$ between π^1 and j^*+1, both P_{σ^j} and $P_{\sigma^1, \ell_b + j - 1}$ consist in scheduling $\{j^*+j, ..., j^*+\ell_e-\ell_b\}\Omega$ starting at time $t_{\pi^j} = \sum_{i \in \Pi^j} p_i$ where Π^j is the jobset of π^j.*

Proof. The first part of the statement follows directly from Definition 9 and simply defines the structure of the children nodes of P_σ. The problem P_{σ^j} is the result of a merging operation with the generic problem $P_{\sigma, \ell_b + j - 1}$ and it could possibly coincide with $P_{\sigma, \ell_b + j - 1}$, for each $j = 1, ..., k-1$. Furthermore, P_{σ^j} is exactly $P_{\sigma, \ell_b + j - 1}$ for $j = k$. The generic structure of $P_{\sigma, \ell_b + j - 1}$ is $\pi\{j^*+1, \ldots, j^*+j-1\}j^*\{j^*+j, \ldots, j^*+\ell_e-\ell_b\}\Omega$, and the merging operations preserve the jobset to schedule after j^*. Thus, we have $\Pi^j = \Pi \cup \{j^*, ..., j^*+j-1\}$ for each $j = 1, ..., k-1$, and this proves the first statement. Analougosly, the structure of $P_{\sigma^1, \ell_b + j - 1}$ is $\pi^1\{j^*+2, \ldots, j^*+j-1\}(j^*+1)\{j^*+j, \ldots, j^*+\ell_e-\ell_b\}\Omega$. Once the subproblem before j^*+1 of size less than k is solved, $P_{\sigma^1, \ell_b + j - 1}$ consists in scheduling the jobset $\{j^*+j, ..., j^*+\ell_e-\ell_b\}$ at time $t_{\pi^j} = \sum_{i \in \Pi^j} p_i$. In fact, we have that $\Pi^j = \Pi^1\{j^*+2, \ldots, j^*+j-1\}\cup\{j^*+1\} = \Pi \cup \{j^*, \ldots, j^*+j-1\}$. ◀

Analogously to the root node, each pair of problems indicated in Proposition 12 can be merged. Again, $(k-1)$ problems P_{σ^j} (with $2 \leq j \leq k$) can be merged with the corresponding

---

Algorithm 2 LEFT_MERGE Procedure

Input: P_σ an input problem of size n, with ℓ_b, j^* accordingly computed
Output: Q: a list of problems to branch on after merging
 1: **function** LEFT_MERGE(P_σ)
 2:     $Q \leftarrow \emptyset$
 3:     **for** $j=1$ to k **do**
 4:         Create P_{σ^j} (j-th child of P_σ) by the improved branching with the subproblem induced by jobset $\{j^*+1, \ldots, j^*+j-1\}$ solved if $\mathcal{L}_\sigma = \emptyset$ or $j=k$
 5:     **end for**
 6:     **for** $j=1$ to $k-1$ **do**
 7:         Create $P_{\sigma^{1j}}$ (j-th child of P_{σ^1}) by the improved branching with the subproblem induced by jobset $\{j^*+2, \ldots, j^*+j-1\}$ solved if $\mathcal{L}_{\sigma^1} = \emptyset$ or $j=k$
 8:         $\mathcal{L}_{\sigma^1} \leftarrow \mathcal{L}_{\sigma^1} \cup \mathrm{BEST}(P_{\sigma^{j+1}}, P_{\sigma^{1j}})$
 9:     **end for**
10:     $Q \leftarrow Q \cup P_{\sigma^1}$
11:     **return** Q
12: **end function**

---

problems P_{σ^1, ℓ_b+j-1}. P_{σ^j} is deleted and P_{σ^1, ℓ_b+j-1} is replaced by P_{σ^1, j^*+j-1} (Figure 3), defined as follows:

- Jobset $\{j^*+j, ..., j^*+\ell_e-\ell_b\}\Omega$ is the set of jobs on which it remains to branch on.
- Let σ^{1, j^*+j-1} be the sequence of positions on which the j^*+j-1 longest jobs $1, ..., j^*+j-1$ are branched, that leads to the best jobs permutation between π^j and $\pi^1\{j^*+2, \ldots, j^*+j-1\}(j^*+1)$ for $2 \le j \le k-1$, and between $\pi\{j^*+1, \ldots, j^*+j-1\}j^*$ and $\pi^1\{j^*+2, \ldots, j^*+j-1\}(j^*+1)$ for $j=k$. This involves the solution of one or two problems of size at most $k-1$ (in $\mathcal{O}^*(2.4143^k)$ time by TTBR) and the finding of the sequence that has the smallest total tardiness value knowing that both sequences start at time 0.

The LEFT_MERGE procedure is presented in Algorithm 2. Notice that this algorithm takes as input one problem and produces as an output its first children nodes to branch on, which replace all its k left-side children nodes.

▶ **Lemma 13.** *The* LEFT_MERGE *procedure returns one node to branch on in $\mathcal{O}(n)$ time and polynomial space. The corresponding problem is of size $n-1$.*

Proof. The creation of problems P_{σ^1, ℓ_b+j-1}, $\forall j = 2, \ldots, k$, can be done in $\mathcal{O}(n)$ time. The call of TTBR costs constant time. The BEST function called at line 8 consists in computing then comparing the total tardiness value of two known sequence of jobs starting at the same time instant: it runs in $\mathcal{O}(n)$ time. The overall time complexity of LEFT_MERGE procedure is then bounded by $\mathcal{O}(n)$ time as k is a constant. Finally, as only node P_{σ^1} is returned, its size is clearly $n-1$ when P_σ has size n. ◀

3.2 Algorithm and complexity analysis

The main procedure TTBM (Total Tardiness Branch-and-Merge) is stated in Algorithm 3. It has a similar recursive structure as TTBR. However, each time a node is opened, the sub-branches required for the merging operations are generated, the subproblems of size less than k are solved and the procedure LEFT_MERGE is called. Then, the algorithm proceeds recursively by extracting the next node from Q with a depth-first strategy and terminates when Q is empty.

▶ **Proposition 14.** *Algorithm TTBM runs in $\mathcal{O}^*(2.247^n)$ time and polynomial space.*

Algorithm 3 Total Tardiness Branch-and-merge (TTBM)

Input: $P : \{1, ..., n\}$: input problem of size n
$\qquad\;\;$ $k \geq 2$: an integer constant
Output: $seqOpt$: an optimal sequence of jobs
1: **function** TTBM(P,k)
2: $\qquad Q \leftarrow P$
3: $\qquad seqOpt \leftarrow$ a random sequence of jobs
4: $\qquad$ **while** $Q \neq \emptyset$ **do**
5: $\qquad\qquad P^* \leftarrow$ extract next problem from Q (depth-first order)
6: $\qquad\qquad$ **if** (the size of $P^* < k$) **then** Solve P^* by calling TTBR
7: $\qquad\qquad$ **end if**
8: $\qquad\qquad$ **if** all jobs $\{1, ..., n\}$ are fixed in P^* **then**
9: $\qquad\qquad\qquad seqCurrent \leftarrow$ the solution defined by P^*
10: $\qquad\qquad\qquad seqOpt \leftarrow$ best solution between $seqOpt$ and $seqCurrent$
11: $\qquad\qquad$ **else**
12: $\qquad\qquad\qquad Q \leftarrow Q \cup \texttt{LEFT\_MERGE}(P^*)$
13: $\qquad\qquad\qquad$ **for** $i = k + 1, ..., n$ **do**
14: $\qquad\qquad\qquad\qquad$ Create child node P_i like in TTBR
15: $\qquad\qquad\qquad\qquad$ **if** P_i is not eliminated by Property 3 **then** $Q \leftarrow Q \cup P_i$
16: $\qquad\qquad\qquad\qquad$ **end if**
17: $\qquad\qquad\qquad$ **end for**
18: $\qquad\qquad$ **end if**
19: $\qquad$ **end while**
20: $\qquad$ **return** $seqOpt$
21: **end function**

Proof. Starting from Algorithm 3, we can derive that for a given problem P of size n, the $(k - 1)$ first children nodes P_2 to P_k are merged with children nodes of P_1. Consequently, among these nodes, only node P_1 remains as a child node of P. For the other $(n - k)$ children nodes, Property 3 is applied eliminating by the way one node over two. The worst-case is achieved when n is odd and k is even and we have the following recurrence:

$$T(n) = T(n - 1) + (T(n - k - 1) + T(k)) + (T(n - k - 3) + T(k + 2)) + ...$$

$$+(T(2) + T(n - 3)) + T(n - 1) + \mathcal{O}(p(n))$$

which can be reformulated as

$$T(n) = 2T(n - 1) + T(n - 3) + ... + T(n - k + 1) + 2T(n - k - 1) + ... + 2T(2) + \mathcal{O}(p(n))$$

Following the same approach used in the proof of Proposition 5, we plug $T(n - 2)$ into the formula and we have

$$T(n) = 2T(n - 1) + T(n - 2) - T(n - 3) + T(n - k - 1) + \mathcal{O}(p(n)) - \mathcal{O}(p(n - 2))$$

The solution of this recurrence is $T(n) = \mathcal{O}^*(c^n)$ with c the largest root of

$$1 = \frac{2}{x} + \frac{1}{x^2} - \frac{1}{x^3} + \frac{1}{x^{k+1}}$$

When k is large enough, the last term in the equation can be ignored, leading to a value of c which tends towards 2.24698 as k increases. More concretely, TTBM runs in $\mathcal{O}^*(2.247^n)$ when $k \geq 14$. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\blacktriangleleft$

4 Conclusions

In this paper an exact exponential algorithm for the single machine total tardiness problem was provided. By exploiting some inherent properties of the problem, we first

proposed a branch-and-reduce algorithm, denoted by TTBR running in $\mathcal{O}^*(2.4143^n)$ time and polynomial space. This algorithm is then improved by means of a merging technique leading to a time complexity $\mathcal{O}^*(2.247^n)$ and polynomial space. The resulting algorithm is named branch-and-merge. The merging technique is shown here on left-side branches only. However, at the price of a very long and technical study, also merging right-side branches can be considered leading to a general branch-and-merge algorithm converging to a $\mathcal{O}^*(2^n)$ worst-case time complexity (and still polynomial in space). The presentation of the right-side merging operation is omitted here due to paper length limitation. A complete description can be found in [5].

As a future development of this work, our aim is twofold. First, we aim at applying the branch-and-merge approach to other combinatorial optimization problems in order to establish its potential generalizability. Second, we want to explore the practical efficiency of branch-and-merge for the single machine total tardiness problem and check whether the merging mechanism and related memorization techniques may improve in practice the performances of known approaches such as the one in [11].

References

1 Federico Della Croce, R Tadei, P Baracco, and A Grosso. A new decomposition approach for the single machine total tardiness scheduling problem. *Journal of the Operational Research Society*, pages 1101–1106, 1998.

2 Jianzhong Du and Joseph Y-T Leung. Minimizing total tardiness on one machine is NP-hard. *Mathematics of Operations Research*, 15(3):483–495, 1990.

3 Hamilton Emmons. One-machine sequencing to minimize certain functions of job tardiness. *Operations Research*, 17(4):701–715, 1969.

4 Fedor V Fomin and Dieter Kratsch. *Exact exponential algorithms.* Springer Science & Business Media, 2010.

5 Michele Garraffa, Lei Shang, Federico Della Croce, and Vincent T'Kindt. An Exact Exponential Branch-and-Merge Algorithm for the Single Machine Total Tardiness Problem. submitted to TCS, 2017. URL: `https://hal.archives-ouvertes.fr/hal-01477835`.

6 Christos Koulamas. The single-machine total tardiness scheduling problem: review and extensions. *European Journal of Operational Research*, 202(1):1–7, 2010.

7 Eugene L Lawler. A "pseudopolynomial" algorithm for sequencing jobs to minimize total tardiness. *Annals of Discrete Mathematics*, 1:331–342, 1977.

8 Christophe Lenté, Mathieu Liedloff, Ameur Soukhal, and Vincent T'Kindt. Exponential Algorithms for Scheduling Problems. *HAL, https://hal.archives-ouvertes.fr/hal-00944382*, 2014. URL: `https://hal.archives-ouvertes.fr/hal-00944382`.

9 C.N Potts and L.N Van Wassenhove. A decomposition algorithm for the single machine total tardiness problem. *Operations Research Letters*, 1(5):177–181, 1982.

10 Wlodzimierz Szwarc. Single machine total tardiness problem revisited. *Creative and Innovative Approaches to the Science of Management, Quorum Books*, pages 407–419, 1993.

11 Wlodzimierz Szwarc, Andrea Grosso, and Federico Della Croce. Algorithmic paradoxes of the single-machine total tardiness problem. *Journal of Scheduling*, 4(2):93–104, 2001.

12 Wlodzimierz Szwarc and Samar K Mukhopadhyay. Decomposition of the single machine total tardiness problem. *Operations Research Letters*, 19(5):243–250, 1996.

13 Gerhard J. Woeginger. Exact Algorithms for NP-hard Problems: A Survey. In Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi, editors, *Combinatorial Optimization — Eureka, You Shrink!*, volume 2570 of *Lecture Notes in Computer Science*, pages 185–207. Springer Berlin Heidelberg, 2003.